Introduction to Csp-Prover2004

(draft)

Yoshinao Isobe, Markus Roggenbach

Contents

1 Introduction **2**

2 Installing Isabelle2004 **2**

3 Setting up Csp-Prover **2**

4 Starting Csp-Prover **4**

5 Small demonstrations **4**

6 Syntax **6**

7 Domain **10**

8 Semantics **11**

9 Verification **12**

- 9.1 Fixed point induction 12
- 9.2 Expanding 14
- 9.3 Decomposition 16
- 9.4 Internal choice and bind 17

1 Introduction

Csp-Prover is an interactive theorem prover dedicated to refinement proofs within the process algebra Csp. It aims specifically at proofs on infinite state systems, which may also involve infinite non-determinism.

...

2 Installing Isabelle2004

Csp-Prover is encoded in Isabelle2004/HOL-Complex. Isabelle can be installed from the Isabelle Web page:

```
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html
```

For example, download the following files for Linux/x86 from the web page:

```
Isabelle2004.tar.gz
ProofGeneral-3.5.tar.gz
polyml_base.tar.gz
polyml_x86-linux.tar.gz
HOL_x86-linux.tar.gz
HOL-Complex_x86-linux.tar.gz
```

Then, uncompress and unpack them into the /usr/local directory (or your appropriate directory) as follows:

```
tar -C /usr/local -xzf Isabelle2004.tar.gz
tar -C /usr/local -xzf ProofGeneral.tar.gz
tar -C /usr/local -xzf polyml_base.tar.gz
tar -C /usr/local -xzf polyml_x86-linux.tar.gz
tar -C /usr/local -xzf HOL_x86-linux.tar.gz
tar -C /usr/local -xzf HOL-Complex_x86-linux.tar.gz
```

Finally, Isabelle/Isar/HOL is started by

```
/usr/local/Isabelle/bin/isabelle -I HOL
```

and Proof General is started by

```
/usr/local/Isabelle/bin/Isabelle
```

In the rest of this document, we assume that `/usr/local/Isabelle/bin` is an executable path.

3 Setting up Csp-Prover

It is assumed that Csp-Prover2004-2.tar.gz has been downloaded from

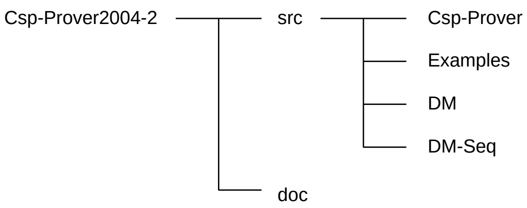


Figure 1: The directory tree of Csp-Prover2004-2

```
http://www...
```

and has been unpacked to the directory (or your appropriate directory):

```
~/tool/Csp-Prover2004-2
```

The directory Csp-Prover2004-2 has the tree shown in Figure 1. Each directory is used as follows:

- `Csp-Prover` contains the theory files for CSP-Prover
- `Examples` contains small examples for testing CSP-Prover.
- `DM` contains the theory files for an example called Dining Mathematicians.
- `DM-Seq` is another version of Dining Mathematicians. A sequential behavior Seq equivalent to Sys is given between Spc and Sys.
- `doc` contains documents for CSP-Prover.

We recommend you to make a heap file: `Csp-Prover`, although you can directly load `Csp_Prover.thy`, and then prove it, and then use it. If you make the heap file once, you do not prove them again before using them. The heap file is made as follows:

1. Go to the directory `Csp-Prover` by

   ```
   cd ~/tool/Csp-Prover2004-2/src/Csp-Prover
   ```

2. Make a heap file `Csp-Prover` by

   ```
   isatool usedir -b HOL-Complex Csp-Prover
   ```

   The heap file will be made in your isabelle directory. If you did not specify the directory, it is probably

   ```
   ~/isabelle/heaps/polyml-*** (which depends on your OS)
   ```

   It may take time to make the heap file. For example, 7 minutes by Pentium M (1.5GHz).

In addition, if you like to comfortably read theory files of `Csp-Prover` by browsers (e.g. Netscape, mozilla, $\cdots$), you can make html files for them as follows:

1. Go to the directory `src` by

   ```
   cd ~/tool/Csp-Prover2004-2/src
   ```

2. Make html-files by

   ```
   isatool usedir -i true HOL-Complex Csp-Prover
   ```

3. Browse theory files and theory dependency-graphs by

   ```
   mozilla ~/isabelle/browser_info/HOL/HOL-Complex/
           Csp-Prover/index.html

   isatool browser ~/isabelle/browser_info/HOL/HOL-Complex/
                   Csp-Prover/session.graph
   ```

   where Java is needed for displaying graphs. The dependency-graph created by isatool for Csp-Prover is shown in Figure 2.

4 Starting Csp-Prover

You can start Csp-Prover in a shell window by

```
isabelle -I Csp-Prover
```

or start it in Proof General by

```
Isabelle -l Csp-Prover
```

We recommend you to use Proof General, which is a superior interface for Isabelle, if possible. Proof General sometimes conflicts your `.emacs` and fails. To avoid this, you may use an option "-u" as follows:

```
Isabelle -u false -l Csp-Prover
```

This option disallows Proof General to use your `.emacs`.

In Proof General, you can also select a logic (e.g. `Csp-Prover`, `HOL`, `HOL-Complex`, $\cdots$) used in Isabelle from the menu bar. Click the button [Isabelle/Isar] → [Logics] → [Csp-Prover].

In addition, you can also activate X-symbols in Proof General from the menu bar. Click the button [Proof General] → [option] → [X-Symbol].

5 Small demonstrations

Try to prove small examples, for getting the outline how Csp-Prover works. If you use a shell window and an editor window, then the proof is proceeding as follows:

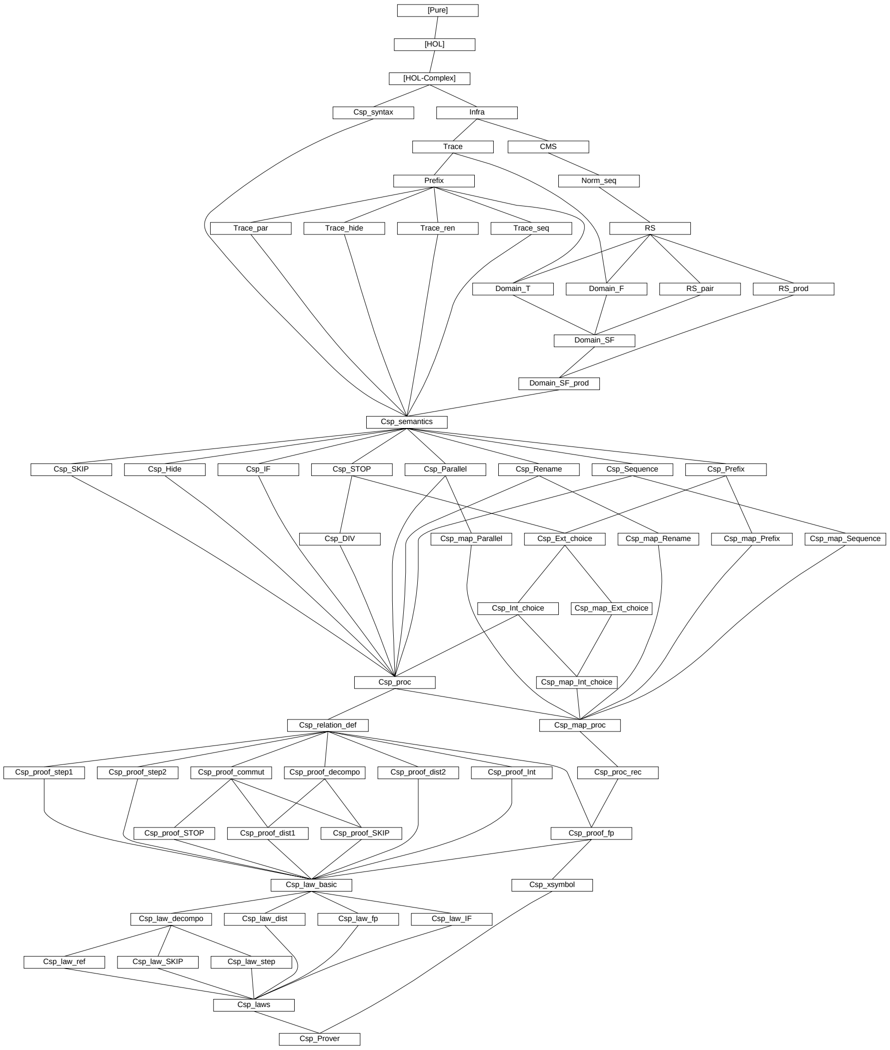


Figure 2: The dependency-graph for Csp-Prover

1. Start Csp-Prover in the shell window by

   ```
   isabelle -I Csp-Prover
   ```

2. Open the following example in the editor window:

   ```
   ~/tool/Csp-Prover2004-2/src/Examples/Inc_nat.thy
   ```

3. Copy the commands from "`Inc_nat.thy`" and paste them to the isabelle window line by line until the proof finishes.

If you can use Proof General, the proof is more elegant as follows:

1. Start Proof General with Csp-Prover by

```
Isabelle -l Csp-Prover
```

2. Open the following example in the Proof General window:

```
~/tool/Csp-Prover2004-2/src/Examples/Inc_nat.thy
```

3. Click the button "Next" in the menu bar until the proof finishes.

More conventional Csp-syntax can be displayed as shown in Figure 3 if you use Proof General and activate X-symbols from the menu bar.

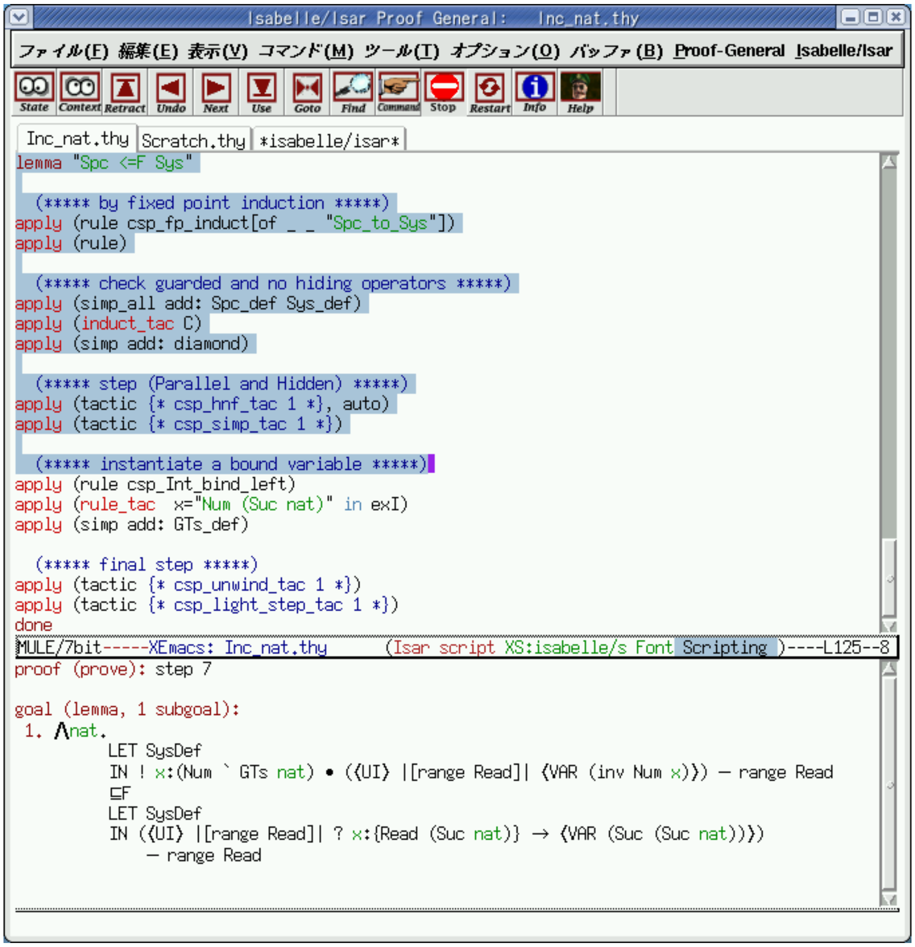
Similarly, try to prove another example:

```
~/tool/Csp-Prover2004-2/src/Examples/Test_Seq.thy
```

6 Syntax

The set of processes are given as a type "`('n,'a) proc`", where "`'n`" and "`'a`" are types of process-names and events, respectively. Each process P whose type is "`('n,'a) proc`" is defined as follows:

```
P ::= SKIP                    %% successful termination
    | STOP                    %% deadlock
    | a -> P                  %% action prefix
    | c ! a -> P              %% sending a over channel c(*)
    | c ? x : X -> P(x)       %% receiving x ∈ X on channel c(*)
    | c ? x -> P(x)           %% receiving x on channel c(*)
    | c !! x : X -> P(x)      %% non-determinisric sending x ∈ X on channel c(*)
    | c !! x -> P(x)          %% non-determinisric sending x on channel c(*)
    | ? x:X -> P(x)           %% external prefix choice
    | ! x:X -> P(x)           %% internal prefix choice (*)
    | P [+] P                 %% external choice
    | P |~| P                 %% internal choice
    | ! x:X .. P(x)           %% internal choice over processes
    | IF b THEN P ELSE P      %% conditional
    | P |[X]| P               %% generalized parallel
    | P ||| P                 %% interleaving (*)
    | P || P                  %% synchronous Parallel (*)
    | P -- X                  %% hiding
    | P [[r]]                 %% relational renaming
    | P ;; P                  %% sequential composition
    | P [> P                  %% (untimed) timeout (*)
    | <C>                     %% process name
```


Isabelle/Isar Proof General: Inc_nat.thy
ファイル(F) 編集(E) 表示(V) コマンド(M) ツール(T) オプション(O) バッファ(B) Proof-General Isabelle/Isar
State
Context
Retract
Undo
Next
Use
Goto
Find
Command
Stop
Restart
Info
Help
Inc_nat.thy
Scratch.thy
isabelle/isar

```
lemma "Spc <=F Sys"

  (***** by fixed point induction *****)
apply (rule csp_fp_induct[of _ _ "Spc_to_Sys"])
apply (rule)

  (***** check guarded and no hiding operators *****)
apply (simp_all add: Spc_def Sys_def)
apply (induct_tac C)
apply (simp add: diamond)

  (***** step (Parallel and Hidden) *****)
apply (tactic {* csp_hnf_tac 1 *}, auto)
apply (tactic {* csp_simp_tac 1 *})

  (***** instantiate a bound variable *****)
apply (rule csp_Int_bind_left)
apply (rule_tac  x="Num (Suc nat)" in exI)
apply (simp add: GTs_def)

  (***** final step *****)
apply (tactic {* csp_unwind_tac 1 *})
apply (tactic {* csp_light_step_tac 1 *})
done
MULE/7bit-----XEmacs: Inc_nat.thy      (Isar script XS:isabelle/s Font Scripting )----L125--8
proof (prove): step 7

goal (lemma, 1 subgoal):
 1. ⋀nat.
          LET SysDef
          IN ! x:(Num ` GTs nat) • ({UI} |[range Read]| ⟨VAR (inv Num x)⟩) — range Read
          ⊑F
          LET SysDef
          IN ({UI} |[range Read]| ? x:{Read (Suc nat)} → ⟨VAR (Suc (Suc nat))⟩)
             — range Read
```

Figure 3: A screen shot of a proof by Csp-Prover

where types of a, X, b, and C are as follows:

a : `'a`
X : `'a set`
b : `bool`
C : `'n`

Alphabetized parallel, future work. Derived operators are marked with (*)

To send or receive values on communication channels, the following short notations are useful:

- Sending a value: "$a\,!\,v$ `->` P" sends a value v to a channel a, and thereafter behaves like P. It is a syntactic sugar of "(a v) `->` P"

- Receiving values: “$a\texttt{?}x\texttt{:}X \texttt{ -> } P(x)$” receives a value x from a channel a if $x \in X$. If X is the universe, “$\texttt{:}X$” can be omitted. They are defined as follows:

$$\begin{array}{rcl} a\texttt{?}x\texttt{:}X \texttt{ -> } P(x) & = & \texttt{?}\ y\texttt{:}\{(a\ x) \mid x \in X\} \texttt{ -> } P(a^{-1}(y)) \\ a\texttt{?}x \texttt{ -> } P(x) & = & a\texttt{?}x\texttt{:UNIV -> } P(x) \end{array}$$

- Non-deterministic Sending a value: “$a\texttt{!!}x \texttt{ -> } P(x)$” non-deterministically sends a value v to a channel a if $v \in X$, and thereafter behaves like $P(v)$. If X is the universe, “$\texttt{:}X$” can be omitted. They are defined as follows:

$$\begin{array}{rcl} a\texttt{!!}x\texttt{:}X \texttt{ -> } P(x) & = & \texttt{!}\ y\texttt{:}\{(a\ x) \mid x \in X\} \texttt{ -> } P(a^{-1}(y)) \\ a\texttt{!!}x \texttt{ -> } P(x) & = & a\texttt{!!}x\texttt{:UNIV -> } P(x) \end{array}$$

Convenient short notations for internal binding are also given as follows:

- Internal Prefix choice: “$\texttt{!}\ x\texttt{:}X \texttt{ -> } P(x)$” requires that an event e can be executed and thereafter it behaves like $P(e)$ for some $e \in X$. It is defined as follows:

$$\texttt{!}\ x\texttt{:}X \texttt{ -> } P(x) \quad = \quad \texttt{!}\ x\texttt{:}X \texttt{ .. } x \texttt{ -> } P$$

- Internal value binding: “$a\texttt{!!}x\texttt{:}X \texttt{ .. } P(x)$” requires that it behaves like $P(v)$ for some $v \in X$, where a is used as a function for translating the type of v to the event type. If X is the universe, “$\texttt{:}X$” can be omitted. They are defined as follows:

$$\begin{array}{rcl} a\texttt{!!}x\texttt{:}X \texttt{ .. } P(x) & = & \texttt{!}\ y\texttt{:}\{(a\ x) \mid x \in X\} \texttt{ .. } P(a^{-1}(y)) \\ a\texttt{!!}x \texttt{ .. } P(x) & = & a\texttt{!!}x\texttt{:UNIV .. } P(x) \end{array}$$

Another short notations for timeout and parallelism are given as follows:

- Timeout: “$P \texttt{ [> } Q''$ behaves like P for a short time before it opts to behave like Q. It is defined as follows:

$$P \texttt{ [> } Q \quad = \quad (P \texttt{ |\textasciitilde| STOP}) \texttt{ [+] } Q$$

- Synchronous Parallel: “$P \texttt{ || } Q$” is a parallel composition, where every event must synchronize between P and Q. It is defined as follows:

$$P \texttt{ || } Q \quad = \quad P \texttt{ |[UNIV]| } Q$$

- Alphabetized Parallel: “$P \texttt{ |[}X\texttt{,}Y\texttt{]| } Q$” is a parallel composition, where P and Q are allowed to communicate by events in X and Y, respectively. It is defined as follows:

$$P \texttt{ |[}X\texttt{,}Y\texttt{]| } Q \quad = \quad P \texttt{ |[}X \cap Y\texttt{]| } Q$$

- Interleaving: "P `|||` Q" is a parallel composition, where P and Q have no communication. It is defined as follows:

$$P \;\texttt{|||}\; Q \;=\; P \;\texttt{|[}\emptyset\texttt{]|}\; Q$$

Next, the set of recursive processes is given as a type "`('n,'a) procRec`". Each recursive process R is defined as follows:

$$R ::= \texttt{LET}\; df \;\texttt{IN}\; P$$

where P is a process whose type is "`('n,'a) proc`" and df has the type "`('n,'a) procDef`" defined as follows:

```
types ('n,'a) procDef = "'n => ('n,'a) proc"
```

Intuitively, the function df defines that a process name C behaves like a process $df(C)$.

Example 6.1 *The recursive process* `Sys` *in* `Inc_nat` *used in Section 5 is defined as follows:*

```
datatype Event = Num nat | Read nat
datatype SysName = UI | VAR nat

consts
  SysDef :: "(SysName, Event) procDef"

primrec
 "SysDef      (UI) = Read?m -> Num!!m -> <UI>"
 "SysDef   (VAR n) = Read!n -> <VAR (Suc n)>"

consts
  Sys :: "(SysName, Event) procRec"

defs Sys_def:
  "Sys == LET SysDef
          IN (<UI> |[range Read]| <VAR 0>) -- (range Read)"
```

□

It is often required to replace each process-name C in a process P with $f(C)$. It is expressed as follows:

$$\texttt{Rewrite}\; P \;\texttt{By}\; f$$

where P and f have types "`('n,'a) proc`" and "`'n => ('m,'a) proc`", respectively. Therefore, `Rewrite` P `By` f is a process, whose type is `('m,'a) proc`, obtained by replacing each process `<C>` with a process $f(C)$.

7 Domain

Csp has a special event `Tick` which means a successful termination. So, the (extended) set of events consists of user-defined events, whose type is `'a`, and `Tick` as follows:

```
datatype 'a event = Ev 'a | Tick
```

In the stable failures model, the behavior of each process is expressed by the order of events that it can perform and a set of refusal events at each state. The order of events is represented by a sequence (i.e. trace) of events defined as follows:

```
typedef 'a trace
   = "{s::('a event) list. Tick ~: set(butlast s)}"
```

where the function `butlast` removes the last element of s and the function `set` transforms a list to a set of elements contained in the list. Therefore, `Tick` does not occur in every trace except the last of the trace.

A failure is a pair (t, X) of a trace and a set of refusal events. Intuitively, a failure (t, X) represents that events included in X cannot be performed after performing t. The type of failures is easily defined as follows:

```
types 'a failure = "'a trace * 'a event set"
```

Finally, the set of domains on the stable failures model $\mathcal{F}$ is given as a type "`'a dom_SF`" in Csp-Prover. The type "`'a dom_SF`" is defined from the set "`'a dom_T`" of traces and the set "`'a dom_F`" of failures as follows:

```
typedef  'a dom_T = "{T::('a trace set). CT1(T)}"

typedef  'a dom_F = "{F::('a failure set). CF1(F)}"

typedef 'a dom_SF = "{SF::('a dom_T * 'a dom_F).
                      CT2(SF) & CF2(SF) & CF3(SF)}"
```

where the conditions are defined as follows:

$$\begin{array}{rcl}
\texttt{CT1}(T) & = & T \neq \emptyset \land \texttt{prefix\_closed}(T) \\
\texttt{CF1}(F) & = & \forall\, s\ X\ Y.\ ((s, X) \in F \land Y \subseteq X) \longrightarrow (s, Y) \in F \\
\texttt{CT2}(T, F) & = & \forall\, s.\ s\texttt{@[Tick]}\ \in T \land \texttt{notick}\ s \\
 & & \quad \longrightarrow (s, \texttt{UNIV-\{Tick\}}) \in F \land (s\texttt{@[Tick]}, \texttt{UNIV}) \in F \\
\texttt{CF2}(T, F) & = & \forall\, s\ X\ Y.\ (s, X) \in F \land \texttt{notick}\ s \\
 & & \quad \land (\forall\, a.\ a \in Y \longrightarrow s\texttt{@[}a\texttt{]} \notin T) \longrightarrow (s, X \cup Y) \in F \\
\texttt{CF3}(T, F) & = & \forall\, s\ X.\ (s, X) \in F \longrightarrow s \in T
\end{array}$$

The set of traces and the set of failures are extracted from a domain on $\mathcal{F}$ by the following functions `Tof` and `Fof`, respectively.

```
consts
  Tof :: "'a dom_SF => 'a dom_T" "Tof SF == fst (Rep_dom_SF SF)"
  Fof :: "'a dom_SF => 'a dom_F" "Fof SF == snd (Rep_dom_SF SF)"
```

where `Rep_dom_SF` is a function which converts the type of "`'a dom_SF`" into `('a dom_T * 'a dom_F)`.

8 Semantics

The meaning of each process P, whose type is "`('n,'a) proc`", is given by translating P to a domain, where it is needed to evaluate each process-name to a domain by a function whose type is "`'n => ('n,'a) proc`". In Csp-Prover 2004, a translation `[[ ]]SF` is a function which takes a process and an evaluation function (for process-names), and then return a pair of a set of traces and a set of failures. In other words, the type of the translation is defined as follows:

```
('n,'a) proc => ('n => 'a dom_SF) => 'a dom_SF
```

The exact definition of `[[ ]]SF` is somewhat complex because it contains many type-conversions by `Rep_dom_*` and `Abs_dom_*`. Here, we show the essence for defining `[[ ]]SF`.

$$\begin{aligned}
\texttt{[[SKIP]]SF} &= \lambda\mathcal{E}.(\{\texttt{[]},\texttt{[Tick]}\},\\
&\qquad\quad \{(\texttt{[]},X) \mid \texttt{Tick} \notin X\} \cup \{(\texttt{[Tick]},X) \mid \texttt{True}\})\\
\texttt{[[STOP]]SF} &= \lambda\mathcal{E}.(\{\texttt{[]}\},\{(\texttt{[]},X) \mid \texttt{True}\})\\
\texttt{[[}a\ \texttt{->}\ P\texttt{]]SF} &= \lambda\mathcal{E}.(a\ \texttt{->t}\ \texttt{Tof}(\texttt{[[}P\texttt{]]SF}(\mathcal{E})), a\ \texttt{->f}\ \texttt{Fof}(\texttt{[[}P\texttt{]]SF}(\mathcal{E})))
\end{aligned}$$

$\cdots$ to be written here soon $\cdots$

$$\texttt{[[<}C\texttt{>]]SF} = \lambda\mathcal{E}.\ \mathcal{E}(C)$$

where

$$\begin{aligned}
a\ \texttt{->t}\ T &= \{\texttt{[]}\} \cup \{\texttt{[Ev}\ a\texttt{]@}s \mid s \in T\}\\
a\ \texttt{->f}\ F &= \{(\texttt{[]},X) \mid \texttt{Ev}\ a \notin X\} \cup \{(\texttt{[Ev}\ a\texttt{]@}s, X) \mid (s,X) \in F\}
\end{aligned}$$

The evaluation for processes is extended for process-definitions (systems of equations) as follows:

```
consts
eval_procDef ::
    "('n => ('m,'a) proc) => ('m => ('m,'a) dom_SF)
                          => ('n => ('n,'a) dom_SF)"  ("[[_]]DF")
defs
  eval_procDef_def :  "[[f]]DF == (λev. (λC. [[f C]]SF ev))"
```

where `%` is the function abstruction symbol (i.e. ASCII symbol for λ).

Finally, recursive processes are evaluated as follows:

```
consts
  FPdom_SF ::"(('n => ('n,'a) dom_SF) => ('n => ('n,'a) dom_SF))
                                       => ('n => ('n,'a) dom_SF)"

  eval_procRec ::"('n,'a) procRec => 'a dom_SF"      ("[[_]]RC")

defs
 FPdom_SF_def :
     "FPdom_SF f == if (∃x. x isUFP f)
                    then (UFP f) else BOTTOM"

primrec
  "[[LET df IN P]]RC = [[P]]SF (FPdom_SF [[df]]DF)"
```

where `FPdom_SF` is a function which evaluate a fixed point of a function `f` if `f` has a unique fixed point.

9 Verification

You can verify the refinement relation `<=F` (whose X-symbol is $\sqsubseteq$`F`) and the equivalence relation `=F`, which are defined as follows, based on the stable failures model in the current CSP-Prover 2004.

$$\begin{aligned} \texttt{R1 <=F R2} &= \texttt{[[R2]]RC} \subseteq \texttt{[[R1]]RC} \\ \texttt{R1 =F R2} &= \texttt{[[R2]]RC} = \texttt{[[R1]]RC} \end{aligned}$$

CSP-Prover gives many CSP-rules for verifying the relations by rewriting CSP-expressions. You can use these CSP-rules in Isabelle by loading the main theory `Csp_Prover`, for example, as follows

```
theory T = Csp_Prover:
```

This means that your theory `T` will be proven by `Csp-Prover`.

In CSP-Prover, proofs mainly consist of three phases: (1) unfolding recursive processes, (2) expanding processes to head-normal-forms (hnf), and (3) decomposing them. These are explained in the rest of this section.

9.1 Fixed point induction

It is hard to verify `(LET df1 IN P1) <=F (LET df2 IN P2)` only by rewriting `P1` and `P2` because they are different processes names defined `df1` and `df2`, respectively. This problem is solved by applying fixed point induction. Therefore, we can verify `(LET df1 IN P1) <=F (LET df2 IN P2)` by proving

```
CHECK (LET df1 IN P1) <=F (LET df2 IN P2) BY f12
```

where it is defined as follows:

```
CHECK (LET df1 IN P1) <=F (LET df2 IN P2) BY f12
=
(∀C. (nohide (df1 C))) ∧ (∀C. (guard (df1 C))) ∧
(∀C. (nohide (df2 C))) ∧ (∀C. (guard (df2 C))) ∧
(LET df2 IN (Rewrite P1 By f12)  <=F LET df2 IN P2 ∧
(∀C. LET df2 IN (Rewrite (df1 C) By f12) C <=F LET df2 IN (f12 C))
```

where `f12` is a function which takes a process-name in `df1` and returns a process-expression containing process-names defined by `df2`, such that for each process name `C`, `f12(C)` refines `C`. It is hard to automatically find such function `f12` from `df1` and `df2`. Therefore, such function will be given by users.

It is important to note that the definition of "`CHECK ...`" contains only process-names defined by `df2` because each process-name `C` defined by `df1` is replaced with `f12(C)`. It allows us to verify the refinement between recursive processes containing different process-names.

For example, the function `Spc_to_Sys` (an instance of `f12` above) which relates `SpcDef` to `SysDef` is defined as follows

```
primrec
  "Spc_to_Sys (Cspc n)
      = (<UI> |[range Read]| <VAR n>) -- (range Read)"
```

in the example `Example/Inc_nat` (also see Section 5 and Example 6.1). Then, when a goal is given as follows:

```
Spc <=F Sys
```

and the following command is applied:

```
apply (rule csp_fp_induct[of _ _ "Spc_to_Sys"])
```

then the following subgoal is returned:

```
CHECK Spc <=F Sys BY Spc_to_Sys
```

This means that we can prove "`CHECK Spc <=F Sys BY Spc_to_Sys`" instead of "`Spc <=F Sys`". The expression "`CHECK ...`" can be unfolded by a command `apply (unfold CHECKref_def)`, however the rule for unfolding it is added to introduction rules which are automatically applied. Thus, just apply the following command,

```
apply (rule)
```

then 6 sub-goals will be displayed in according to the definition of `"CHECK ..."` as follows:

```
goal (lemma (check_ex1), 6 subgoals):
 1. !!C. nohide (LetD Spc C)
 2. !!C. guard (LetD Spc C)
 3. !!C. nohide (LetD Sys C)
 4. !!C. guard (LetD Sys C)
 5.
    LET LetD Sys IN (Rewrite (InP Spc) By Spc_to_Sys) <=F
    LET LetD Sys IN InP Sys
 6. !!C.
      LET LetD Sys IN (Rewrite ((LetD Spc) C) By Spc_to_Sys) <=F
      LET LetD Sys IN Spc_to_Sys C
```

where `LetD R` and `LetP R` are defined as follows:

```
LetD (LET df IN P)  =  df
LetP (LET df IN P)  =  P
```

This strategy (by `csp_fp_induct`) is also available for verification based on equivalence relation `=F`.

9.2 Expanding

To verify `(LET df1 IN P1) <=F (LET df2 IN P2)`, it is useful to transform `P1` and `P2` to head-normal-forms (hnf) such as `? x:X -> P1'` and `? x:Y -> P2'`. To do that, rewriting laws called *step laws* (e.g. see P.32 (1.14) in [Ros98]). are given in Csp. Csp-Prover gives tactics based on step laws for getting hnfs. The most powerful tactic is "`csp_hnf_tac`". This tactic applies step laws Csp-expressions which are unguarded (by Prefix or Prefix choice), unbounded (by Internal bind), and unconditional because of avoiding excessive expanding, which makes expressions to be unreadable. User defined tactics are applied in Isar-mode as follows:

```
apply (tactic {* csp_hnf_tac 1 *})
```

where `1` represents that this tactic is applied to the first subgoal.

The tactic `csp_hnf_tac` contains the following small tactics, and they can be individually applied for reducing proof cost.

- `csp_unwind_tac` is a tactic for unfolding process-names, if every definitions of process-names are guarded and have no hiding.

  For example, assume that a sub-goal is given as follows:

  ```
  Spc <=F
  LET SysDef IN (<UI> |[range Read]| <VAR 0>) -- range Read
  ```

  where `Spc` and `SysDef` are defined in `Inc_nat` (also see Example 6.1). Now, apply the tactic as follows:

```
apply (tactic {* csp_unwind_tac 1 *})
```

Then, the sub-goad will be rewritten to the following new sub-goal:

```
Spc <=F
LET SysDef
IN (? x:range Read -> Num (inv Read x) -> <UI>
     |[range Read]| Read 0 -> <VAR (Suc 0)>)
    -- range Read
```

As shown this result, the unguarded process-names `UI` and `VAR` are replaced with their process-expressions defined by `SysDef`, where short notations for `??` and `!!` are automatically unfolded. It is important that the process-names in the new sub-goal are not unfolded because they are guarded. This technique works for avoiding infinite rewriting.

Note: before applying this tactic, it should be proven that every definitions of process-names are guarded and have no hiding. They are easily proven, for example, by

```
lemma guardSysDef[simp]: "!!C. guard (SysDef C)"
by (induct_tac C, simp_all)
```

but they are not automatically proven by `(auto)`.

- `csp_step_tac` is a tactic for transforming processes of the form `STOP`, `a -> P`, `P [+] Q`, `P |[X]| Q`, `P -- X`, `P [[r]]`, or `P ;; Q`, to processes (hnfs) of the form `? x:A -> P'`, used as follows:

  ```
  apply (tactic {* csp_step_tac 1 *})
  ```

  `csp_light_step_tac` is a tactic for transforming processes of the form `STOP`, or `a -> P`, to processes (hnfs) of the form `? x:A -> P'`. Since the proof cost of `csp_step_tac` is often high, `csp_light_step_tac` is sometimes used instead of `csp_step_tac`.

- `csp_dist_tac` is a tactic for distributing unguarded operators over internal choices and internal binds.

- `csp_simp_tac` is a tactic for simplify processes by mainly evaluating conditions. Simplification rules can be manually added or deleted as follows:

  ```
  apply (tactic {* csp_simp_tac 1 *})
  apply (tactic {* csp_simp_add_tac "name1" 1 *})
  apply (tactic {* csp_simp_del_tac "name2" 1 *})
  apply (tactic {* csp_simp_add_del_tac "name1" "name2" 1 *})
  ```

  where `name1` and `name2` are theory-names added to and deleted from simplification rules, respectively. Do not forget to convert the theory-names to strings by double-quotations.

- `csp_rule_tac` is a tactic for applying an introduction rule to sub-expressions as follows:

```
apply (tactic {* csp_rule_tac "name" 1 *})
```

where `name` is a name of introduction rule.

- `csp_asm_tac` is a tactic for applying assumptions.

9.3 Decomposition

It is possible to decompose process-expressions to sub-expressions and verify the sub-expressions because of their congruency and monotonicity.

- `csp_rm_head` is a rule for removing the same head of head-normal forms like

  ```
  LET df IN ? x:X -> P =F LET df IN ? x:Y -> Q
  ```

  Since this is added to (automatically applied) introduction rules, it can be easily applied as follows:

  ```
  apply (rule)
  ```

  Then, the above sub-goal is decomposed to 2 subgoal:

  ```
  1. X = Y
  2. !!a. a : Y ==> LET df IN P =F LET df IN Q
  ```

  The first goal may be solved by Isabelle set-theory and a tactic like `csp_hnf_tac` can be applied to the second goal again, to transforms them to hnfs.

- `csp_decompo` is more powerful rule than `csp_rm_head`, and it decomposes expresstions if their outermost operators are the same. For exapmle, the goal

  ```
  LET df IN P1 [+] P2 =F LET df IN Q1 [+] Q2
  ```

  is decomposed the two sub-goals

  ```
  1. LET df IN P1 =F LET df IN Q1
  2. LET df IN P2 =F LET df IN Q2
  ```

  by the proof command

  ```
  apply (rule csp_decompo)
  ```

  However, this rule is unsafe because unexpected decomposition can be done. For example, do not apply this rule to the goal

  ```
     LET df IN a -> SKIP [+] b -> SKIP
  =F LET df IN b -> SKIP [+] a -> SKIP
  ```

- `csp_free_decompo` is a rule for decomposing expressions if they are not the forms of `a -> P`, `? x:X -> P`, `! x:X .. P`, and `IF b THEN P ELSE Q`, thus only unguarded expressions (by events or condistions) can be decomposed. This rule can be used for avoiding excessive decompositions, which are often caused by `csp_decompo`.

- `csp_decompo_subset` is a rule for removing internal binds. Thus, the goal

  ```
  LET df IN ! x:X .. P(x) <=F LET df IN ! x:Y .. Q(x)
  ```

  is rewitten to the following sub-gaols

  ```
  1. Y <= X
  2. !!a. a:Y ==> LET df IN P(a) <=F LET df IN Q(a)
  ```

  by the proof command

  ```
  apply (rule csp_decompo_subset)
  ```

  This rule also works for the expressions whose operators are internal and external prefix choices as follows:

  ```
  LET df IN ! x:X -> P(x) <=F LET df IN ? x:Y -> Q(x)
  ```

  This goal is rewitten to the following sub-gaols by this rule:

  ```
  1. Y ~={}
  2. Y <= X
  3. !!a. a:Y ==> LET df IN P(a) <=F LET df IN Q(a)
  ```

9.4 Internal choice and bind

References

[Ros98] A.W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.